METHOD AND SYSTEM FOR SYNTHESIZING A CIRCUIT REPRESENTATION INTO A NEW CIRCUIT REPRESENTATION HAVING GREATER UNATENESS

GOVERNMENT RIGHTS

This invention was made with government support under National Science Foundation Grant Nos. 9503463 and 9872066 and DARPA Grant No. DABT 63-96-C-0074. The government has certain rights in this invention.

CROSS-REFERENCE TO RELATED APPLICATION

This application claims the benefit of U.S. provisional application Serial No. 60/226,103, filed August 17, 2000 and entitled "Method and System for Synthesizing Digital Circuits with Unateness Properties."

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to a method and system for synthesizing a circuit representation of a circuit into a new circuit representation having greater unateness.

2. Background Art

The number of transistors that can be fabricated in a single IC has been growing exponentially over the last three decades. A well-known example is the Intel series of microprocessors. Intel's first commercial microprocessor, the 4004, was built with 2,300 transistors in 1971, whereas a recent Intel microprocessor, the Pentium III, introduced in 1999 contains 9.5 million transistors. The clock frequency of the microprocessors also has dramatically increased from the 4004's 0.1 MHZ to the Pentium III's 550MHz. The 1998 International Technology

Roadmap for Semiconductors developed by the Semiconductor Industry Association (SIA) predicts that the transistor count and the clock frequency of ICs will grow even faster in the next decade.

It is thus becoming extremely difficult and time-consuming to design all the components in a complex IC from scratch, verify their functional and timing correctness, and ensure that overall performance requirements are met. To solve this challenging problem, a "design reuse" methodology is being widely introduced, which integrates large standardized circuit blocks into a single IC called a system on a chip (SOC). An SOC integrates a set of predesigned "off-the-shelf" blocks to build the entire system on a single chip, just as off-the-shelf IC's have been used to build a system on a board. The SOC methodology allows IC designers to focus on the interfaces linking the predesigned blocks. Thus it saves a tremendous amount of time that the designers would have spent creating all the blocks from scratch, and verifying their correctness. For this reason, the SOC design approach is becoming increasingly popular.

A large portion of the reused blocks in an SOC are intellectual property (IP) circuits, which are also called cores or virtual components, and are often provided by third party vendors. The IP providers typically transfer their designs to SOC designers in a way that hides the key design details of the IP circuits and so protect the IP provider's investment in the IP designs. The IP circuits that have been developed so far cover numerous functions and support many different IC technologies. Additionally, the number and scope of the available IP circuits are rapidly growing.

IP circuits are currently available in three different forms known as hard, soft, and firm. Hard IP circuits are provided in the form of complete layouts that are optimized and verified by the IP providers for a particular IC technology. Therefore, hard IP circuits can save time in all SOC design steps, but cannot be re-optimized by system designers for other technologies. The intellectual property of hard IP circuits includes all the implementation details and is protected by providing the system designers only with the circuit's high-level behavioral or functional

specifications. Soft IP circuits are provided in the form of register-transfer level (RTL) descriptions, which define the circuit's behavior using a set of high-level blocks. These blocks can be converted by system designers to lower-level designs at the gate and physical levels. Thus soft IP circuits can be optimized for a variety of IC technologies and performance requirements, while they can save SOC design time in the high-level design and verification steps. Their RTL designs are considered to be the intellectual property contents of soft IP circuits. Finally, firm IP circuits are provided as netlists or gate-level descriptions. They allow the system designers to optimize the IP circuit's physical design such as cell placement and routing for various IC technologies. They provide the major advantages of both hard and soft IP circuits—they save system design time while allowing the flexibility of retargeting the IP circuits at various IC technologies. Both their RTL and gate-level designs are considered to be the intellectual property contents of firm IP circuits. While hard IP circuits are primarily aimed at ASIC designs, some soft and firm IP circuits are aimed at SOCs implemented using field programmable gate array (FPGA) technology.

Although the advancement of IC technology allows extremely large designs to be integrated in a single chip, today's IC technology presents major challenges to the existing design and testing methodologies. For example, testing requirements for complex digital circuits are becoming increasingly tighter. Using traditional processes to synthesize the implementation of digital and other circuits often leads to circuits that are either inefficient in meeting testing requirements (i.e., unreasonably large test sets, etc.) or cannot satisfy design constraints (i.e., delay, area limits, etc.).

The unateness of a circuit has a substantial impact on circuit testability and performance. Unate variables of a circuit representation z are variables that appear only in complemented or uncomplemented form in z's minimal two-level expressions such as sum of products (SOP) or product of sums (POS) expressions; binate variables are non-unate. For example, $z_1 = a\bar{b} + a\bar{c} + bcd$ is a minimal SOP expression for the four-variable function z_1, in which a and d are unate, and b and c are binate.

The majority of circuit representations are, in nature, binate, and it is often difficult to synthesize these binate functions into a circuit implementation that can be efficiently tested for manufacturing defects and operate at very high speeds.

For example, a high-speed circuit implementation known as "domino logic" requires that the circuit to be implemented be unate. Therefore, binate circuit functionality to be implemented in domino logic must be decomposed into unate circuit implementations. Similarly, static CMOS logic implementations become efficient if unate circuit implementations are extracted from an original binate circuit prior to implementation. Datapath logic circuits such as adders, subtractors, comparators, and ALUs are good examples of applications where carry generation functions (i.e., unate functions) are extracted from a larger (often binate) function and implemented in high-speed circuit structures.

Another advantage of unate circuit implementations is their relatively small universal test set (i.e., the set of minimal true and maximal false test vectors for a function z). These test vectors have the useful property that they can detect all multiple stuck-at faults in any implementation of z. Universal test sets guarantee a very high coverage of manufacturing defects in a vast range of implementations for given circuit functionality. The universal test sets for binate circuit implementations tend to become excessively large. In addition, the unateness property enables the generation of test vectors from the behavioral functions of circuits before their implementations are actually executed.

Existing functional decomposition processes are not suited to the goal of decomposing a binate circuit representation into a small set of unate subfunctions. One existing functional decomposition process called kernel extraction is aimed at multi-level logic synthesis. The kernels of a circuit representation f are defined as f's cube-free primary divisors or quotients. For example, the kernels of $f = (a + b + c)(d + e)f + bfg + h$ include $d + e$, $d + e + g$, and $a + b + c$. This decomposition process employs algebraic division operations with the kernels serving as divisors or quotients to f. Here algebraic division represents f by a logic

expression of the form $f = p \cdot q + r$. The kernels of f can be binate, so f's subfunctions p, q, and r can also be binate. In addition, kernel extraction often leads to an excessive number of subfunctions, and so is not practical for unate decomposition.

Another existing decomposition process is Boole-Shannon expansion, which represents circuit implementation f by $xf_x + \bar{x}f_{\bar{x}}$ where f_x is the cofactor of f with respect to x. Cofactor f_x is defined as the subfunction of f obtained by assigning 1 to variable x in f. Boole-Shannon expansion has been widely used for binary decision diagram (BDD) construction, technology mapping and field programmable gate array synthesis. Boole-Shannon expansion is unsuited to the goal of obtaining a small set of unate circuit implementations, however, since it may only make the child functions f_x and $f_{\bar{x}}$ unate, while always expressing the parent function in a binate form. When applied repeatedly, Boole-Shannon expansion can also produce an unnecessarily large number of subfunctions, as each is created by eliminating only one binate variable at a time.

Finally, a disjoint or disjunctive decomposition represents a boolean function $f(X, Y)$ in the form $h(g_1(X_1), g_2(X_2), \ldots, g_k(X_k), Y)$, where X_1, X_2, ..., X_k, and Y are disjoint (i.e., non-overlapping) variable sets. This decomposition is relatively easy to compute, and is sometimes used for logic synthesis problems where the interconnection cost dominates the circuit's overall cost. It has the drawback that many circuit representations cannot be disjointly decomposed, and, like Boole-Shannon expansion, it can make the parent function h binate. Thus, the disjoint decomposition technique is also not appropriate for our unate decomposition goal.

SUMMARY OF THE INVENTION

One object of the present invention is to provide a method and system for efficiently synthesizing a circuit representation into a new circuit representation (i.e., circuit implementation) having greater unateness. Notably, those of ordinary skill in the relevant art will appreciate that the present invention may be

implemented or applied in a variety of circumstances to synthesize circuits beyond those discussed, by way of example, in the preceding Background Art.

One advantage of circuit implementations that possess unateness is their relatively small universal test set (i.e., the set of its minimal true and maximal false test vectors). Universal test sets guarantee a very high coverage of manufacturing defects in a vast range of implementations for a give function. Unlike universal test sets for highly unate circuit implementations, the universal test sets for largely binate circuits tend to become excessively large. In addition, the unateness property enables the generation of test vectors from the behavioral functions of circuits before their implementations are actually executed.

Another advantage of unate circuit implementations is their low chip area. Yet another advantage of unate circuits is their ability to operate at very high speeds. For example, a low-area high-speed circuit implementation known as "domino logic" requires that the boolean function to be implemented be unate. Therefore, binate functions to be implemented in domino logic must be decomposed into unate functions prior to implementation. Similarly, static CMOS logic implementations become efficient if unate functions are extracted from an original binate function prior to implementation. Datapath logic circuits such as adders, subtractors, comparators, and ALUs are good examples of applications where carry generation functions (i.e., unate functions) are extracted from a larger (often binate) function and implemented in high-speed circuit structures.

To meet these and other objects and advantages of the present invention, a method having preferred and alternate embodiments is provided for synthesizing a representation of a circuit into a new representation having greater unateness. The method includes (i) partitioning a circuit representation to obtain a representation of at least one sub-circuit, (ii) recursively decomposing the representation of the at least one sub-circuit into a sum-of-products or product-of-sums representation having greater unateness than the representation of the at least one sub-circuit, and (iii) merging the sum-of-products or product-of-sums representation into the circuit representation to form a new circuit representation.

The invented method may additionally include repeating steps (i), (ii) and (iii) until a desired level of unateness for the new circuit representation has been achieved.

The invented method may additionally include, for each decomposition, selecting the sum-of-products or product-of-sums representation having fewer binate variables.

The invented method may additionally include merging common expressions of the sum-of-products or product-of-sums representations.

The invented method may additionally include implementing algebraic division to merge common unate expressions.

The invented method may additionally include partitioning the circuit representation to obtain a representation of at least one sub-circuit that is highly unate.

The invented method may additionally include implementing a binary decision diagram to recursively decompose the representation of the at least one sub-circuit into the sum-of-products or product-of-sums representation. The binary decision diagram may be a zero-suppressed binary decision diagram.

Additionally, a system having preferred and alternate embodiments is provided for synthesizing a circuit representation into a new circuit representation having greater unateness. The system comprises a computing device configured to (i) receive input defining a circuit representation, (ii) partition the circuit representation to obtain a representation of at least one sub-circuit, (iii) recursively decompose the representation of the at least one sub-circuit into a sum-of-products or product-of-sums representation having greater unateness than the representation of the at least one sub-circuit, (iv) merge the sum-of-products or product-of-sums representation into the circuit representation to form the new circuit representation, and (v) output the new circuit representation.

The computing device may be further configured to receive input defining a desired level of unateness for the new circuit representation, and repeat steps (ii), (iii) and (iv) until the desired level of unateness is achieved.

The computing device may be further configured for each decomposition, select the sum-of-products or product-of-sums representation having fewer binate variables.

The computing device may be further configured to merge common expressions of the sum-of-products or product-of-sums representations.

The computing device may be further configured to implement algebraic division to merge common expressions.

The computing device may be additionally configured to partition the circuit representation to define a representation of at least one sub-circuit that is highly unate.

The computing device may be additionally configured to implement a binary decision diagram to recursively decompose the representation of the at least one sub-circuit into a sum-of-products or product-of-sums representation. The binary decision diagram may be a zero-suppressed binary decision diagram.

The circuit representation and the new circuit representation may be input to the computing device and output from the computing device, respectively, in a hardware description language such as Verilog or VHDL.

Additionally, a preferred computer-readable storage embodiment of the present invention is provided. In accord with this embodiment, a computer-readable storage medium contains computer executable code for instructing one or more computers to (i) receive input defining a circuit representation, (ii) partition the circuit representation to obtain a representation of at least one sub-circuit, (iii) recursively decompose the representation of the at least one sub-circuit into a sum-

of-products or product-of-sums representation having greater unateness than the representation of the at least one sub-circuit, (iv) merge the sum-of-products or product-of-sums representation into the circuit representation to form a new circuit representation, and (v) output the new circuit representation.

The computer executable code may additionally instruct the computer(s) to receive input defining a desired level of unateness for the new circuit representation, and repeat steps (ii), (iii) and (iv) until the desired level of unateness is achieved.

The computer executable code may additionally instruct the computer(s) to, for each decomposition, select the sum-of-products or product-of-sums representation having fewer binate variables.

The computer executable code may additionally instruct the computer(s) to merge common expressions of the sum-of-products or product-of-sums representations.

The computer executable code may additionally instruct the computer(s) to implement algebraic division to merge common expressions.

The computer executable code may additionally instruct the computer(s) to employ a binary decision diagram to recursively decompose the representation of the at least one sub-circuit into the sum-of-products or product-of-sums representation.

The above objects and advantages of the present invention are readily apparent from the following detailed description of the preferred and alternate embodiments, when taken in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURES 1a-1c are functional circuit diagrams illustrating (a) a single binate block with three functions, (b) two unate blocks with six functions, and (c) three unate blocks with fifteen functions;

FIGURE 2 is a block representation corresponding to a decomposition of $f(X) = h(g_1(X), g_2(X), \ldots, g_k(X))$ in accordance with the present invention;

FIGURES 3a-3b are (a) a binary tree representing an AND-OR decomposition in accordance with the present invention and (b) a corresponding block representation in accordance with the present invention;

FIGURE 4 is a block representation corresponding to a *k*-level AND/OR tree in accordance with the present invention;

FIGURE 5 is an example gate-level circuit to be decomposed by the computer-implemented process *UDSYN* in accordance with the present invention;

FIGURES 6a-6i are (a) a circuit graph G_{C1} for the circuit of Figure 5, (b) partition G_{P1} selected from G_{C1}, (c) unate decomposition performed on G_{P1}, (d) graph G_{C12} obtained by merging B_2 and G_{C1}, (e) partition G_{P2} selected from G_{C2}, (f) final unate decomposition of G_{P2}, (g) circuit graph G_{C3} obtained by merging B_3 and G_{C2}, (h) partition G_{P3} selected from G_{C3}, and (i) final block representation for the circuit of Figure 5 in accordance with the present invention;

FIGURES 7a-7c illustrate a zero-suppressed binary decision diagram (BDD) representing $f_4^{SOP} = ab + \bar{c}\bar{b}$: (a) a path representing cube ab; (b) a path representing cube $\bar{c}\bar{b}$; and (c) a zero-suppressed BDD representing $f_4^{POS} = (a + \bar{c})(a + \bar{b})(b + \bar{c})$ in accordance with the present invention; and

FIGURES 8a-8d illustrate four steps in partitioning G_{C1} of Fig. 6*a*: (a) start with a primary input node n10 and select n7; (b) select n14 and its transitive fanin nodes; (c) select n18 and its transitive fanin nodes; (d) final partition G_{P1} in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention comprises a method and system having preferred and alternate embodiments for efficiently synthesizing a representation of a circuit into a new representation having greater unateness.

For purposes of illustration, the present invention is described in the context of Bodean function-based digital circuitry. However, application of the present invention is not so limited. Notably, the present invention may be applied to a variety of circuit representations such as gate-level circuits, PLA representation, transistor-level representations, and HDL-based circuits.

Fig. 1*a* defines a circuit representation 10 with three binate outputs (e.g., x, y and z). Figure 1*b* shows a two-block representation for circuit 10, and Fig. 1*c* a three-block representation. The block representations of (b) and (c) are obtained by decomposing the functions defining the original single-block circuit representation 10. Circuit functionality for each block is represented in a sum-of-product (SOP) or product-of-sum (POS) form. Table 1 compares the test requirements for the three representations of Fig. 1.

TABLE 1

Block Representation	*Test Requirements*		*Implementation Flexibility*	
	No. of Binate Variables for all Functions	Universal Test Set Size	No. of Block Functions	Area of an Example Synthesized Circuit
Figure 1a	7	64	3	28
Figure 1b	0	33	6	29
Figure 1c	0	47	12	50

In the case of Fig. 1*a*, function *z* is binate for all six variables, so its universal test set consists of all 64 possible test vectors. The decomposed designs of Figs. 1*b* and *c* require smaller universal test sets (33 and 47, respectively), since all their internal blocks are unate. Although the difference in test set size is minor in this small example, it tends to be significant for larger circuits.

Circuits that do not have natural block representations are often implemented by logic synthesis systems, while those with natural block representations are often implemented manually. Although the present invention is not limited to the former case, we assume that the target circuit is of the former type in order to best describe the present invention.

For a given circuit, the block representations created in accordance with the present invention can be considered as design constraints. In other words, the boundaries of the blocks serve as a high-level structural design constraint that must be satisfied by low-level implementations of the circuit such as gate-level or transistor-level implementations. Notably, these design constraints tend to restrict implementation flexibility.

The outputs of any blocks in a circuit *C*'s block representation S_B define the block functions. Circuit *C*'s implementation flexibility can be roughly

measured by the number of block functions that C employs. This follows from the fact that a large number of block functions in S_B implies that the functions themselves are small. Block functions in different blocks cannot be merged, so implementations of small block functions are generally less flexible than large ones. Thus the fewer the block functions in S_B, the higher the implementation flexibility of C.

For example, Table 1 compares the implementation flexibility of the block representations in Fig. 1. Figure 1*b* has three more block functions than Fig. 1*a*, while Fig. 1*c* has 12 more. Whereas Fig. 1*a* has full implementation flexibility, the other block representations have limited flexibility. The last column of Table 1 compares the area of some example implementations synthesized by a commercial synthesis system Synopsys® Design Compiler with the goal of reducing area. The area is calculated from the relative gate areas defined in the Synopsys cell library; for example, inverter = 1, AND2 = 2, AND4 = 3, OR2 = 2, and OR4 = 3. In summary, this example suggests that lower implementation flexibility often leads to poor implementations in terms of circuit area.

To permit a broad range of implementation styles, the invented synthesis process attempts to decompose a binate function into as small a set of unate subfunctions as possible. In general, a decomposition of function f can be expressed as:

$$f(X) = h(g_1(X), g_2(X), \ldots, g_k(X))$$

Let f be the root function, subfunction h the parent function, and each subfunction g_i a child function. $X = \{x_1, x_2, \ldots, x_n\}$ denotes the support set of f and each g_i. A decomposition is called a *unate decomposition* if all the subfunctions h and $g_{1:k}$ in $f(X)$ are unate. A decomposition of the form $f(X)$ transforms a single-block model of function f into a two-block model with h defining one block and $g_{1:k}$ the other; see Figures 2 and 3.

In accordance with the present invention, a preferred method for synthesizing circuits utilizing unateness properties of boolean functions involves recursive OR and AND decompositions of a circuit output function f and its subfunctions. The OR and AND decompositions represent $f(X)$ by $h(g_1(X), g_2(X))$, where h has the form $g_1 + g_2$ and $g_1\ g_2$, respectively. We obtain an OR decomposition of f from an SOP form of f, and an AND decomposition from a POS form.

For example, consider a binate function f_1 whose SOP form is $a\bar{b} + \bar{a}c + \bar{c}a$. A possible OR decomposition of f_1 is $h = g_1 + g_2$ with $g_1 = \bar{a}c$ and $g_2 = a\bar{b} + \bar{c}a$. This decomposition makes all the subfunctions h, g_1, and g_2 unate, and so is a unate decomposition. Now consider f_1's POS form $(a + c)(\bar{a} + \bar{b} + \bar{c})$. We can obtain directly from this form an AND decomposition with subfunctions $h = g_1\ g_2$, $g_1 = a + c$, and $g_2 = \bar{a} + \bar{b} + \bar{c}$. This is also a unate decomposition.

A single OR or AND decomposition of a large binate function may not lead to a unate decomposition. However, a sequence of OR or AND decompositions applied to f recursively always produces a unate decomposition for any function f. The general form of such a sequence with k levels is

$$f = h^1(g_1^1, g_2^1)$$
$$g_i^1 = h^2(g_1^2, g_2^2)$$
$$\vdots$$
$$g_i^{k-1} = h^k(g_1^k, g_2^k)$$

where h^j and g_i^j denote a parent function and a subfunction, respectively, produced by the j-th level decomposition. Parent function h^j can be either AND or OR. A k-level sequence of AND and OR decompositions forms a binary AND-OR tree, where the internal nodes represent AND or OR operations, while the leaf nodes denote unate subfunctions that need no further decomposition.

An arbitrary sequence of AND and OR decompositions can lead to an excessive number of subfunctions. To reduce this number, we restrict our attention to sequences of the following type, which we refer to as *unate AND-OR decompositions*.

$$f = h^1(g_u^1, g_b^1)$$
$$g_b^1 = h^2(g_u^2, g_b^2)$$
$$\vdots$$
$$g_b^{k-1} = h^k(g_u^k, g_b^k)$$

As in the general case, h^j is either AND or OR, but the final g_b^k and every g_u^j are unate, while every g_b^j except the final one is either unate or binate. This decomposition can also be represented in the compact factored form

$$f = h^1(g_u^1, h^2(g_u^2, h^3(g_u^3, \ldots, h^{k-1}(g_u^{k-1}, h^k(g_u^k, g_b^k))\ldots))) \qquad (1)$$

as well as the general form

$$f(X) = h(g_u^1, g_u^2, \ldots, g_u^k, g_b^k). \qquad (2)$$

Comparing (1) with (2), we see that the parent function h in (2) is composed of the AND and OR subfunctions $h^1, h^2, \ldots, h^k$ only, and so is always unate.

Figure 3*a* shows a binary AND-OR tree corresponding to the unate decomposition

$$f = (g_u^1 \cdot (g_u^2 + (g_u^3 + (g_u^4 \cdot (g_u^5 \cdot g_b^5)))))$$

obtained by a unate AND-OR decomposition with 5 levels. The internal nodes 30a-30e in Fig. 3a represent AND or OR operations, while the leaf nodes 32a-32f at the bottom represent unate subfunctions. Figure 3*b* depicts the block representation corresponding to Figure 3a. B_1 defines an AND-OR tree network that implements

the function h. B_2 is a network of undefined internal representation that implements the unate subfunctions g_u^1, g_u^2, g_u^3, g_u^4, g_u^5, and g_b^5.

In general, we obtain the foregoing kind of unate AND-OR decomposition for f as follows: first decompose f into g_u^1 and g_b^1 using an AND or OR operation that makes g_u^1 unate; then repeatedly decompose g_b^j into g_u^{j+1} and g_b^{j+1} in a similar way, until g_b^{j+1} becomes unate. This must eventually happen, because a g_b^j of a single product or sum term is unate. In practice, the AND-OR decomposition process often terminates with a final g_b^j consisting of a relatively large unate function.

As noted above, the global parent function $h(g_u^1, \ldots, g_u^k, g_b^k)$ in (2) is unate. Thus, the final result of a k-level AND-OR decomposition is a set of k + 2 unate subfunctions $g_u^{1:k}$, g_b^k, and $h(g_u^1, \ldots, g_u^k, g_b^k)$.

Notably, an important goal of the block synthesis method shown in Figure 3 is to find an AND-OR decomposition of a given function f using as few subfunctions as possible. In addition, it is preferred that each g_u^j be selected in a manner that makes the resulting g_b^j highly unate. This selection often leads to a unate decomposition involving few subfunctions. Also such a g_u^j can be relatively easily derived from a standard SOP or POS form.

Each level of a unate AND-OR decomposition is defined by either an AND or OR operation. How we select the operation at each level has a large impact on the final result, as we show with the following example.

Consider $f_2 = a \oplus b \oplus c$ whose SOP and POS forms are given below.

$$f_2^{SOP} = \bar{a}\bar{b}c + a\bar{b}\bar{c} + \bar{a}b\bar{c} + abc \tag{3}$$

$$f_2^{POS} = (\bar{a} + b + c)(a + \bar{b} + c)(a + b + \bar{c})(\bar{a} + \bar{b} + \bar{c}) \tag{4}$$

OR decompositions are derived from (3), and AND decompositions are derived from (4). Suppose we select an OR operation in every level of the decomposition. A possible result is:

$$f_2 = g_u^1 + (g_u^2 + (g_u^3 + g_b^3))$$

which involves five unate subfunctions:

$$g_u^1 = \bar{a}\bar{b}c\,;$$
$$g_u^2 = \bar{a}b\bar{c}\,;$$
$$g_u^3 = abc\,;$$
$$g_b^3 = abc\,;\text{ and}$$
$$h(g_u^1, g_u^2, g_u^3, g_b^3)\,.$$

Note that in this particular example, the unate decomposition is completed when the final g_b^k is a product term, and the resulting g^j subfunctions correspond to each of the product terms in (3).

Next, suppose we select an AND operation in every level. A possible result is the unate decomposition $f_2 = g_u^1 \cdot (g_u^2 \cdot (g_u^3 \cdot g_b^3))$, which involves five subfunctions:

$$g_u^1 = \bar{a} + b + c\,;$$

$$g_u^2 = a + \bar{b} + c\,;$$

$$g_u^3 = a + b + \bar{c}\,;$$

$$g_b^3 = \bar{a} + \bar{b} + \bar{c}\,;\text{ and}$$

$$h(g_u^1, g_u^2, g_u^3, g_b^3)\,.$$

Notably, a unate decomposition of f_2 can be obtained involving fewer subfunctions if AND and OR operations are mixed as follows. Suppose we select an OR operation in the first level and an AND operation in the second level. The OR operation decomposes (3) into $f_2^1 = g_u^1 + g_b^1$, where $g_u^1 = \bar{a}\bar{b}\bar{c}$ and $g_b^1 = a\bar{b}\bar{c} + \bar{a}b\bar{c} + abc$. To apply an AND operation to g_b^1, we use g_b^1's POS form $(a+b)(a+\bar{c})(b+\bar{c})(\bar{a}+\bar{b}+c)$. Then an AND operation leads to $g_b^1 = g_u^2 \cdot g_b^2$, where for example, $g_u^2 = (\bar{a}+\bar{b}+\bar{c})$ and $g_b^2 = (a+b)(a+\bar{c})(b+\bar{c})$. Since g_b^2 is unate, the unate decomposition is complete. Note that unlike the previous cases, the final g_b^k here contains more than one term. The third unate decomposition of f_2 is

$$f_2 = g_u^1 + (g_u^2 \cdot g_b^2) = \bar{a}\bar{b}\bar{c} + (\bar{a}+\bar{b}+\bar{c}) \cdot (a+b)(a+\bar{c})(b+\bar{c})$$

which involves only four subfunctions, one less than the first and second cases, where we selected three AND and three OR operations, respectively. This example shows that how we select the AND-OR operation in each level of the AND-OR decomposition is very important.

Often, there are many possible AND and OR decompositions in each level. This implies the existence of a large number of possible unate AND-OR decompositions. For example, if an SOP form of g_b^j contains m product terms, we can partition these terms into two groups defining g_u^{j+1} and g_b^{j+1} in 2^m different ways. At each level, either an AND or OR operation can be chosen, so the number of possible k-level unate AND-OR decompositions is 2^{m+k}. Thus, finding a unate AND-OR decomposition of a large function f involving a minimal number of subfunctions is often impractical. We therefore introduce *Unate-Decomp,* a heuristic process that systematically selects AND or OR decompositions at each recursion level, and produces a final unate AND-OR decomposition containing relatively few subfunctions.

Unate-Decomp represents a function f and all its subfunctions in both SOP and POS forms. To produce an AND (OR) decomposition of f, it selects a set

S of product terms (sum terms) from the SOP (POS) form of f, so that S constitutes a unate subfunction g_u. The rest of the product terms (sum terms) of f define subfunction g_b. *Unate-Decomp* then represents g_b by both SOP and POS forms, which it uses to produce an OR and AND decomposition at the next recursion level. To represent SOP and POS forms efficiently, binary decision diagrams can be employed.

To decompose f into as few unate subfunctions as possible, *Unate-Decomp* produces each g_u^j in a way that reduces the number of binate variables in g_b^j. In the case of multiple-function circuits, *Unate-Decomp* first decomposes each output function f_i using the method described above. It then merges common subfunctions of different functions f_i and f_j to reduce the total number of subfunctions.

Notably, representing large circuits directly by two-level expressions is often inefficient. To handle such cases efficiently, a preferred process first partitions a given circuit, and then performs decomposition on each partition. For example, one partition is created for each process of *Unate-Decomp* parent function h. The resulting decomposition is then merged into the rest of the circuit. Then the next partition is created from the merged circuit, and the next process of *Unate-Decomp* is conducted. This process is repeated until no more partitioning is necessary.

Preferably, each decomposition step and circuit partition are selected in a way that produces a small number of highly unate subfunctions. Consequently, the resulting block representations tend to have a high level of implementation flexibility.

To decompose f into as few unate subfunctions as possible, *Unate-Decomp* produces each g_u^j in a way that reduces the number of binate variables in g_b^j. For example, consider the following function f_3:

$$f_3 = \bar{a}b\bar{c} + \bar{a}d + \bar{a}\bar{e} + b\bar{c}f + bdf + b\bar{e} + \bar{c}df + \bar{c}\bar{e} + a\bar{b}d\bar{f} + a\bar{b}\bar{e} + acdf + c\bar{d}f + cd + \bar{d}e$$

Suppose we decompose f_3 into $g_u + g_b$. Table 2 shows some possible ways of doing this and the number of binate variables in the resulting g_b.

TABLE 2

g_u	g_b	No. of Binate Variables in g_b
$cd + acdf + bdf$	$\bar{a}d + a\bar{b}e + b\bar{c}f + c\bar{d}f + \bar{a}b\bar{c} + a\bar{b}d\bar{f} + \bar{a}\bar{e} + \bar{d}e + cdf + b\bar{e} + ce$	6
$a\bar{b}d\bar{f} + \bar{c}\bar{e}$	$\bar{a}d + a\bar{b}e + b\bar{c}f + c\bar{d}f + \bar{a}b\bar{c} + \bar{a}\bar{e} + \bar{d}e + \bar{c}df + b\bar{e} + cd + acdf + bdf$	5
$b\bar{c}f + bdf + \bar{c}df + b\bar{e} + \bar{c}\bar{e} + \bar{a}b\bar{c} + \bar{a}\bar{e}$	$cd + acdf + a\bar{b}e + c\bar{d}f + \bar{a}d + a\bar{b}d\bar{f} + \bar{d}e$	3
$\bar{a}b\bar{c} + \bar{a}d + \bar{a}\bar{e} + b\bar{c}f + bdf + b\bar{e} + \bar{c}df + \bar{c}\bar{e}$	$a\bar{b}d\bar{f} + a\bar{b}e + acdf + c\bar{d}f + cd + \bar{d}e$	2

While the first OR decomposition produces g_b with six binate variables, the last OR decomposition produces g_b with only two binate variables, and so is selected.

In each level of the decomposition process, *Unate-Decomp* produces a pair of AND and OR decompositions using a special form of cofactor operation called *Subset*. The *Subset* operation for a literal l_i extracts a subset S of product (sum) terms of a given SOP (POS) form by eliminating terms that contain l_i. For example, applying *Subset* to the SOP form

$$\bar{a}bc + a\bar{c}d + a\bar{b}d$$

for literal

$$\bar{a}$$

yields

$$S = a\bar{c}d + a\bar{b}d$$

Unate-Decomp systematically computes S for a set of binate literals $\{l_i\}$ so that S is unate and the set of other terms is highly unate. Then S defines g_u^j, and the other product terms define g_b^j.

After a unate decomposition is formed, *Unate-Decomp* constructs two blocks from an AND-OR tree representing the decomposition; see Fig. 4. To ensure that all the block functions are unate, we place in block B_1 all the nodes representing the subfunctions $g_u^{1:k}$ and g_b^k, which correspond to the leaf nodes in the AND-OR decomposition tree. We place in block B_2 all the other nodes, which represent AND and OR operations and together form the function h.

In the preceding description, we focused on decomposing a single function. In the case of multiple-function circuits, *Unate-Decomp* first decomposes each output function f_i using the method described above. It then merges common subfunctions of different functions f_i and f_j to reduce the total number of subfunctions. Algebraic division operations are often employed by logic synthesis techniques to efficiently combine common expressions. These operations can be easily applied to the results of our AND-OR decompositions, and often reduce the number of subfunctions significantly. Notably, *Unate-Decomp* incorporates

algebraic division in such a manner that two different functions share the divisor of each division.

Based on the unate decomposition concept described above, we introduce a computer-implemented synthesis program in accordance with the present invention called *Unate Decomposition Synthesis* (UDSYN). Representing large circuits directly by two-level expressions is often inefficient. To handle such cases efficiently, UDSYN first partitions the given circuit, and then performs decomposition on each partition, as generally described above.

Table 3 contains one embodiment of a pseudocode representation of *UDSYN* in accordance with the present invention. Notably, it is understood by those of ordinary skill in the art that different computer programs and program arrangements can be implemented to support and execute the overall function of *UDSYN*.

TABLE 3

Embodiment of process *UDSYN* (Verilog-input)

```
1:  G_C := Build-Circuit-Graph(Verilog-input);
2:  while (G_C ≠ Ø) begin
3:    G_P := UD-Partition(G_C);                  /* G_P is a graph representing a partitioned block */
4:    G_C := G_C - G_P;                          /* Remove nodes in G_P from G_C */
5:    for each output node n_R in G_P begin
6:          Build-ZSBDD(n_R, G_P);               /* Create SOP and its complement for n_R */
7:    end;
8:    (B_i, B_{i+1}) := Unate-Decomp(G_P);       /* B_i and B_{i+1} correspond to B_1 and B_2 of Fig. 7 */
9:    G_C := G_C ∪ B_{i+1}; i := i + 1;          /* Insert notes in B_{i+1} into G_C */
10:  end;
11: Verilog-output := Interconnect-Blocks({B_i}); /* Verilog-output is the final block representation */
12: return Verilog-output;
```

UDSYN takes an input circuit in a form such as a Verilog specification whose internal elements can be either functional descriptions or gates.

First, *UDSYN* builds a circuit graph G_C whose nodes represent the internal elements. It then creates a partition G_P of G_C using *UD-Partition*(G_C), and removes nodes in G_P from G_C. The output functions of G_P are represented in SOP and POS forms. The process *Unate-Decomp*(G_P) performs unate AND-OR decompositions on the output nodes in G_P, and constructs decomposed blocks B_1 and B_2 as in Fig. 4. Blocks B_1 and B_2 created from the i-th partition G_{Pi} are denoted by B_i and B_{i+1}, respectively. Step 9 modifies G_C by inserting all nodes of B_2 into G_C. *UDSYN* repeats the above steps until all nodes in G_C are removed. It then constructs a hardware description language (e.g. Verilog, VHDL, etc.) output file by specifying the interconnections among all blocks B_i.

We illustrate UDSYN using a gate-level circuit of Fig. 5 as input. Figures 6*a* to *i* show intermediate results of steps 2 to 10 in Table 3. Figure 6*a* shows the circuit graph G_{C1} for the circuit of Fig. 5; each node in G_{C1} corresponds to a gate in the circuit. *UD-Partition*(G_C) creates a partition G_{P1} starting from the primary inputs of G_{C1}. The shading in G_{C1} indicates nodes that are selected by *UD-Partition*(G_C), and constitute G_{P1}. Figure 6*b* represents G_{P1} by a rectangle. All nodes in G_{P1} are removed from G_{C1}, and are merged into SOP and POS forms by steps 5 to 8. These SOP and POS forms are decomposed by step 8 into unate subfunctions; these subfunctions are grouped into two blocks B_1 and B_2 as in Fig. 4. As Fig. 6*c* shows, B_1 consists of seven subfunctions and B_2 consists of three subfunctions. We create a new circuit graph G_{C2} by merging B_2 and G_{C1} as shown in Fig. 6*d*. Returning to step 3, *UD-Partition*(G_C) selects some nodes (shaded) in G_{C2} and creates a new partition G_{P2}, which is represented by a rectangle in Fig. 6*e*. Then step 8 decomposes G_{P2} into blocks B_2 and B_3 appearing in Fig. 6*f*. Step 9 merges B_3 into a new circuit graph G_{C3} as in the preceding steps; see Fig. 6*g*. Figure 6*h* shows a new partition G_{P3} constructed from G_{C3}. By repeating this process, we finally obtain the decomposed block representation of Fig. 6*i* consisting of five blocks $B_{1:5}$. The output functions of these blocks are described by Verilog code in equation form. If *UD-Partition*(G_C) constructs k partitions, UDSYN produces a total of $k + 1$ blocks.

Step 6 of Table 3 uses a type of binary decision diagram (BDD) called a zero-suppressed BDD (ZSBDD) to represent the SOP and POS forms of functions. Although other forms of BDD can be used, we limit our attention in this description to ZSBDDs for the sake of presentation. A ZSBDD of a function f is a graph whose paths denote the product terms (cubes) in an SOP form of f. UDSYN uses two ZSBDDs to represent a pair of SOP and POS forms for the internal function of each node in an AND-OR tree like that in Fig. 4. Thus an AND-OR tree with n nodes is represented by $2n$ individual ZSBDDs, each of which is linked to the corresponding node in the tree.

For example, Figs. 7*a* and *b* show a ZSBDD representing $f_4^{SOP} = ab + \bar{c}\bar{b}$. The internal nodes (circles) in Figs. 7*a* and *b* denote the literals appearing in f_4^{SOP}. The terminal or leaf nodes (rectangles) denote the output values that f_4^{SOP} generates when its literals are set to the values specified on the edges. The name "zero-suppressed" stems from the property that all nodes in a ZSBDD whose 1-edge points to the 1-terminal node are eliminated. Every path from the root to the 1-terminal node represents a cube in f_4^{SOP}. For example, the path highlighted by the dashed line in Fig. 7*a* represents cube *ab*, while the one highlighted in Fig. 7*b* represents cube $\bar{c}\bar{b}$. Although ZSBDDs can represent only SOP forms directly, POS forms can also be handled by their complemented form.

For example, consider the POS expression

$$f_4^{POS} = (a + \bar{c})(a + \bar{b})(b + \bar{c})$$

having the following complement

$$\overline{f_4^{POS}} = \bar{a}c + \bar{a}b + \bar{b}c$$

Figure 7*c* shows a ZSBDD that represents $\overline{f_4^{POS}}$, where every path from the root to the 1-terminal node represents a sum term in f_4^{POS} with their literals complemented. In this way, we can represent both SOP and POS forms using ZSBDDs.

ZSBDDs have been shown to represent large functions efficiently. This is due to the fact that small ZSBDDs can contain a large number of paths, so a large number of cubes can be often represented by a compact ZSBDD. ZSBDDs also support fast manipulation of sets of cubes such as finding subsets, computing the intersection of cube sets, and performing a type of division operation on cube sets. Utilizing these features, we implement unate AND-OR decomposition and division processes that act directly on ZSBDDs.

UD-Partition(G_C) creates a partition of the input circuit in a way that makes the functions defining the partition highly unate, while meeting a specified partition size limit. Partitions created in this way simplify the unate decomposition process. One pseudo-code embodiment of *UD-Partition* appears in Table 4. Notably, it is understood by those of ordinary skill in the art that a variety of different computer programs and program arrangements can be implemented to support and execute the inventive function of *UD-Partition*.

TABLE 4

Embodiment of process *UD*-Partition(G_C)

1: **for each** n_c in G_C in level order **begin**
2: **for each** fan in node n_i of n_c **begin**
3: **if** (n_i is a primary input of G_C) **then**
4: $S_{supp}(n_c) := S_{supp}(n_c) \cup n_i$; /* $S_{supp}(n_c)$ is n_c's support set */
5: **else**
6: $S_{supp}(n_c) := S_{supp}(n_c) \cup S_{supp}(n_i)$;
7: ***Calculate-Path-Count***($S_{supp}(n_c)$); /* Add path count of fan-in nodes to n_c */
8: **end**;
9: $N_{BV}(n_c) = \sum_{i=1}^{k} Binate(s_i, n_c)$ /* *Binate*(s_i, n_c) = 1 if s_i is binate for n_c */
10: **end**;
11: **while** ($G_C \neq \emptyset$) **begin**
12: $n_m := $ ***Select-Node-of-Min-***$N_{BV}(G_C)$; /* n_m is to be included in the partition */
13: $S_N :=$ nodes in n_m's fan-in cone in G_C;
14: **if** (***I/O-count***($G_P \cup S_N$) < threshold) **then** /* G_P is the graph for the partition */
15: $G_P := G_P \cup S_N$; $G_C := G_C - G_P$; /* Add n_m and its transitive fan-in nodes to G_P */
16: **else break**; /* Discard the candidate node n_m */
17: **end**;
18: **return** (G_C, G_P);

Steps 1 to 10 compute the number of binate support variables of each node in the circuit graph G_C. Steps 11 to 17 create the current partition G_P by selecting nodes in G_C that have a minimal number of binate variables.

A node n_c in G_C is unate with respect to a primary input s_i, if all paths between n_c and s_i have the same inversion parity; otherwise, n_c is binate with respect to s_i. To determine the inversion parity of the paths, we calculate the number of paths from the primary inputs to each node n_c in G_C. Let $p_{even}(s_i, n_c)$ and $p_{odd}(s_i, n_c)$ be the number of paths from a primary input s_i to a node n_c whose inversion parity is even and odd, respectively. Steps 3 to 7 find the set $S_{supp}(n_c)$ of support variables for each node n_c. For n_c and its fanin nodes n_i, Calculate-Path-Count obtains $p_{even}(s_i, n_c)$ and $p_{odd}(s_i, n_c)$ by recursively computing

$$p_{even}(s_i, n_c) = p_{even}(s_i, n_c) + p_{even}(s_i, n_i)$$
$$p_{odd}(s_i, n_c) = p_{odd}(s_i, n_c) + p_{odd}(s_i, n_i)$$

if the inversion parity from n_i to n_c is even; otherwise, it computes

$$p_{even}(s_i, n_c) = p_{even}(s_i, n_c) + p_{odd}(s_i, n_i)$$
$$p_{odd}(s_i, n_c) = p_{odd}(s_i, n_c) + p_{even}(s_i, n_i).$$

The binary function $Binate(s_i, n_c)$ produces 1 (0), if a node n_c is binate (unate) with respect to its support variable s_i, and is computed by

$$Binate(s_i, n_c) = 0, \text{ if } p_{even}(s_i, n_c) = 0 \text{ or } p_{odd}(s_i, n_c) = 0$$
$$Binate(s_i, n_c) = 1, \text{ otherwise}$$

The number $N_{BV}(n_c)$ of binate variables of node n_c with k variables is defined as

$$N_{BV}(n_c) = \sum_{i=1}^{k} Binate(s_i, n_c)$$

The intuition behind using $N_{BV}(n_c)$ to guide the partitioning stems from the fact that the more binate the node n_c, the more difficult the decomposition process for n_c tends to be. Steps 1 to 10 traverse every node only once, which has complexity $O(N)$. They propagate $p_{even}(s_i, n_c)$ and $p_{odd}(s_i, n_c)$ for every s_i for a n_c to the nodes in the transitive fanout of n_c, which also accounts for complexity $O(N)$. Hence the overall complexity of computing $N_{BV}(n_c)$ for all nodes in G_C is $O(N^2)$.

For example, Table 5 shows the calculation of $N_{BV}(n_c)$ for every node in Fig. 6*a*.

TABLE 5

Node n_c	No. of paths from n_c's support variable s_i s_i, $P_{even}(s_i,n_c)$, $P_{odd}(s_i,n_c)$)	No. of binate variables $N_{BV}(n_c)$
*n*5	(*a*, 0, 1), (*b*, 0, 1)	0
*n*10	(*b*, 0, 1), (*d*, 0, 1)	0
*n*3	(*f*, 1, 0), (*d*, 1, 0)	0
*n*12	(*g*, 0, 1)	0
*n*13	(*h*, 1, 0), (*i*, 1, 0)	0
*n*8	(*c*, 0, 1), (*b*, 1, 0), (*d*, 1, 0)	0
*n*7	(*g*, 1, 0), (*f*, 0, 1), (*d*, 0, 1)	0
*n*14	(*h*, 1, 0), (*i*, 1, 0), (*f*, 1, 0), (*d*, 1, 0)	0
*n*9	(*c*, 1, 0), (*b*, 0, 1), (*d*, 0, 1)	0
*n*11	(*b*, 1, 1), (*d*, 1, 1), (*c*, 1, 0), (*e*, 0, 1)	2
*n*4	(*c*, 1, 1), (*b*, 1, 1), (*d*, 2,2), (*g*, 1, 1), (*f*, 1,1)	5
*n*2	(*c*, 0, 1), (*b*, 2, 0), (*d*, 1, 0), (*a*, 1, 0)	0
*n*6	(*a*, 1, 1), (*b*, 2, 2), (*c*, 1, 1), (*d*, 1, 1)	4
*n*18	(*c*, 1, 1), (*b*, 1, 1), (*d*, 3, 2), (*g*, 1, 1), (*f*, 2, 1), (*h*, 1, 0), (*i*, 1, 0)	5
*n*15	(*c*, 2, 2), (*b*, 4, 4), (*d*, 3, 3), (*a*, 1, 1), (*e*, 1, 1)	5
*n*1	(*f*, 0, 1), (*d*, 1, 2), (*a*, 1, 1), (*b*, 2, 2), (*c*, 1, 1)	4

Node n_c	No. of paths from n_c's support variable s_i s_i, $P_{even}(s_i,n_c)$, $P_{odd}(s_i,n_c)$)	No. of binate variables $N_{BV}(n_c)$
$n16$	(f, 3, 3), (d, 7, 7), (a, 2, 2), (b, 6, 6), (c, 4, 4), (g, 2, 2)	6
$n17$	(g, 5, 5), (f, 7, 7), (d, 15, 15), (a, 4, 4), (b, 12, 12), (c, 8, 8)	6

The second column lists $p_{odd}(s_i, n_c)$ and $p_{even}(s_i, n_c)$ computed for each node n_c and all its support variables. The last column gives $N_{BV}(n_c)$. For example, for $n_c = n11$, *Binate*(b, $n11$) = 1, *Binate*(d, $n11$) = 1, *Binate*(c, $n11$) = 0, and *Binate*(e, $n11$) = 0. Thus $N_{BV}(n_c) = 1 + 1 + 0 + 0 = 2$.

After $N_{BV}(n_c)$ is computed for every n_c in G_C, *UD-Partition* selects from G_C a node n_m of minimal $N_{BV}(n_c)$ starting from a primary input of G_C. It then inserts into G_P all non-partitioned nodes in the transitive fanin region of n_m. This process is repeated until the size of G_P exceeds a threshold equal to the maximum number of G_P's I/O lines. By limiting the partition size in this way, we can prevent ZSBDDs from exploding for large circuits, while producing a partition with highly unate output functions.

Figure 8 illustrates how we partition G_C of Fig. 6*a*. Suppose we limit G_P's I/O lines to seven inputs and six outputs, that is, we set the threshold to 7/6. The $N_{BV}(n_c)$ values calculated in Table 5 are shown next to each node n_c in Fig. 8. Figures 8*a* to *d* indicate the current G_P created in each iteration by shading, and newly selected nodes by thick circles. The first n_m is selected from the candidate nodes $n3$, $n5$, $n8$, n10, $n11$, $n12$, and $n13$, which are adjacent to the primary inputs. We select $n3$ whose $N_{BV}(n_c)$ has the minimum value 0, and add it to G_P; see Fig. 8*a*. The next search begins from $n3$ and selects $n3$'s fanout node $n7$ whose $N_{BV}(n_c)$ = 0. We then select all nodes in the transitive fanin region of $n7$; Fig. 8*a* indicates these selected nodes by a dashed line. Figure 8*b* shows the current G_P consisting of $n3$, $n12$, and $n7$. We then select $n14$ over $n1$, $n4$, and $n17$, and then select $n14$'s fanin node $n13$; the newly selected nodes are again indicated by a dashed line in Fig. 8*b*. We next select $n17$ over $n18$, $n1$, $n4$, but $n17$ leads to a partition with

seven outputs, one greater than the limit six. Hence we select $n18$ instead which has the next smallest $N_{BV}(n_c)$. We then select nodes in $n18$'s transitive fanin region; see Fig. 8*c*. At this point, the number of I/O lines of G_P equals the threshold 7/6, so the partitioning is done. Figure 8*d* indicates the final G_P by shading.

Since *UD-Partition* selects nodes with fewer binate variables first, it often leads to a partition where many output functions are already unate and so require no further decomposition. For example, in G_P of Fig. 8*d*, four output functions at $n3$, $n7$, $n8$, and $n10$ are unate. Figure 6*c* shows a unate decomposition of this G_P, where nodes $g3$, $g7$, $g8$, and $g10$ in D_1 correspond to these four unate functions, and so are not decomposed.

Next we describe *Unate-Decomp*(G) which systematically applies unate AND-OR decomposition operations to a circuit partition. See Table 6 for one pseudo-code embodiment of *Unate-Decomp*(G) .

TABLE 6

Embodiment of process *Unate-Decomp*(G)

```
1:  S_B := G's binate function nodes;          /* S_B stores nodes of binate functions to be decomposed */
2:  while (S_B ≠ Ø) begin
3:      for each node n_B in S_B begin
4:          (n_u, n_b) := ANDOR-OneLevel(G, n_B);     /*n_u (n_b) points to subfunction g_u (g_b) in (3.4) */
5:          S_D := S_D ∪ {n_u, n_b};                  /* S_D stores candidate divisor nodes */
6:      end;
7:      for each node n_d in S_D begin
8:          for each node n_f in S_B - S_D begin       /* n_f is a candidate dividend node */
9:              (n_q, n_r) := Division(n_f, n_d);     /* n_q is the quotient and n_r is the remainder */
10:             if (N_BV(n_f) < N_BV(n_q)+N_BV(n_r)) then
11:                 Reverse the division;
12:         end;
13:     end;
14:     S_B := Ø; S_D := Ø;
15:     for each node n_i in G begin                  /* Find new nodes to be decomposed */
16:         if (N_BV(n_i) > threshold)
17:             S_B := S_B ∪ n_i;                     /* n_i exceeds the threshold */
18:     end;
19: end;
20: return G;
```

Graph G initially contains the nodes in the current partition. Steps 3 to 6 perform a level of AND-OR decomposition on every binate node n_B in G. Then, steps 7 to 13 perform division operations on every binate node in G by treating as divisors child nodes created by the AND-OR decompositions. $N_{BV}(n_i)$ denotes the number of binate variables in the subfunction at node n_i in G. If a division reduces $N_{BV}(n_i)$, it is accepted; otherwise, it is discarded. The above process is repeated until all nodes in G become unate. For some large binate functions, forcing all nodes to be unate leads to an excessive number of subfunctions. We therefore stop decomposing a node n_i if $N_{BV}(n_i)$ becomes less than a certain threshold. This threshold is chosen to yield a small set of subfunctions at the cost of lower unateness. Thus the threshold allows us to trade the level of unateness for a higher implementation flexibility of the block representation.

Table 7 contains a pseudo-code embodiment of the computer-implemented process *ANDOR-OneLevel*(G, n_B), which implements one level of the AND-OR decomposition technique described earlier.

TABLE 7

Embodiment of process *ANDOR-OneLevel*(G, n_B)

1: (g_u^{SOP}, g_b^{SOP}) := ***Find-Unate-Cube-Set*** ($SOP(n_B)$); /* OR decomposition */

2: (g_u^{CPOS}, g_b^{CPOS}) := ***Find-Unate-Cube-Set*** ($Inv(SOP(n_B))$); /* AND decomposition */

3: **if** $(N_{BV}(g_b^{SOP}) \leq N_{BV}(g_b^{CPOS}))$ **then**

4: ***Replace-Node*** (n_B, ***NewNodes*** $(h^{SOP}, g_u^{SOP}, g_b^{SOP})$) ; /* Replace n_B in G by the new nodes */

5: **return** $(Node(g_u^{SOP}), Node(g_b^{SOP}))$;

6: **else**

7: ***Replace-Node*** $(n_B, NewNodes(h^{POS}, Inv(g_u^{CPOS}), Inv(g_b^{CPOS})))$; /* $Inv(g)$ complements g */

8: **return** $(Node(Inv(g_u^{CPOS})), Node(Inv(g_b^{CPOS})))$;

The process *Find-Unate-Cube-Set*($SOP(n_B)$) finds a set of unate cubes (product terms) from an SOP representation $SOP(n_B)$ for node n_B. This operation forms an OR decomposition. An AND decomposition is obtained by complementing the input $SOP(n_B)$ and the outputs of *Find-Unate-Cube-Set,*

respectively. This enables ZSBDDs to handle both AND and OR decompositions, although ZSBDDs can only represent SOP forms directly.

Table 8 contains a pseudo-code embodiment of the computer-implemented process *Find-Unate-Cube-Set(ISOP)*.

TABLE 8

```
Embodiment of process Find-Unate-Cube-Set (ISOP)        /* ISOP is the initial SOP form */
1:   S_best := SOP := ISOP;
2:   while (N_BV(SOP) > threshold) begin
3:       for each literal l_i for binate variables in SOP repeat
4:           S_i := Subset(SOP, l_i);                   /* Remove all cubes containing l_i */
5:           if (N_BV(S_i)+N_BV(ISOP - S_i) < N_BV(S_best)+N_BV(ISOP - S_best)) then
6:               S_best := S_i;
7:       end;
8:       SOP := S_best;
9:   end;
10:  g_u := S_best;
11:  g_b := ISOP - S_best;
12:  return (g_u, g_b);
```

Find-Unate-Cube-Set(ISOP) derives a cube set S from f's initial SOP form *ISOP* so that S meets the threshold on $N_{BV}(S)$. As a result, S defines unate subfunction g_u^k in (1), while *ISOP* - S defines g_b^k. As discussed earlier, an exact method to find an optimum AND-OR decomposition of an m-term *ISOP* must examine up to 2^m possible AND decompositions. To avoid this and derive S efficiently, a type of cofactor operation is implemented which can simultaneously extract from *ISOP* multiple cubes (product terms) with a common property. This operation, denoted by *Subset(SOP, l_i)*, removes from *SOP* all cubes that contain literal l_i. Thus l_i does not appear in the resulting SOP form S_i, while $\bar{l}_i$ may appear in S_i; hence S_i is unate with respect to l_i.

For example, consider a function f_5 whose SOP form is

$$SOP = ab\bar{c} + a\bar{c}d + a\bar{d} + \bar{b}\bar{d}a + ed$$

Subset(*SOP*, $\bar{d}$) removes cubes $a\bar{d}$ and $\bar{b}\bar{d}a$ which contain literal $\bar{d}$, and yield

$$S_i = ab\bar{c} + a\bar{c}d + ed$$

The basic concept is to apply *Subset*(*SOP*, l_i) to a set of binate literals in a way that makes both S_i and *SOP* - S_i highly unate. We found that for a binate l_i, *Subset*(*SOP*, l_i) often eliminates from *SOP* not only l_i but also other binate literals. Hence we can often obtain a highly unate cube set S by repeating only a few steps of *Subset*(*SOP*, l_i). The inner loop (steps 3 to 8) of Table 8 performs *Subset*(*SOP*, l_i) for all binate literals l_i, and selects S_{best}, (i.e., the S_i having the minimum $N_{BV}(S_i) + N_{BV}(ISOP - S_i)$). The outer loop (steps 2 to 9) repeats this process recursively with S_{best} in place of *SOP* until $N_{BV}(S_{best})$ becomes less than *threshold*. The final S_{best} defines g_u, while *ISOP*-S_{best} defines g_b.

To illustrate, consider an initial SOP form

$$ISOP = ab\bar{c} + a\bar{c}d + a\bar{d} + \bar{b}\bar{d}a + ec + \bar{a}bde + ed$$

Suppose that the threshold of N_{BV} is 0. Table 9 shows each step of the outer loop in its first iteration with *ISOP* assigned to *SOP*.

TABLE 9

$SOP = ISOP = ab\bar{c} + a\bar{c}d + a\bar{d} + \bar{b}\bar{d}a + ec + \bar{a}bde + ed$			
Binate literal l_i	S_i	$ISOP - S_i$	$N_{BV}(S_i) + N_{BV}(ISOP - S_i)$
a	$\bar{b}c + \bar{a}bde + ed$	$ab\bar{c} + a\bar{c}d + a\bar{d} + \bar{b}\bar{d}a$	3
$\bar{a}$	$ab\bar{c} + a\bar{c}d + a\bar{d} + \bar{b}\bar{d}a + \bar{b}c + ed$	$\bar{a}bde$	3
b	$a\bar{d} + \bar{b}\bar{d}a + ed + \bar{b}c$	$ab\bar{c} + a\bar{c}d + \bar{a}bde$	2
$\bar{b}$	$ab\bar{c} + a\bar{c}d + a\bar{d} + ed + \bar{a}bde$	$\bar{b}\bar{d}a + \bar{b}c$	2
c	$ab\bar{c} + a\bar{c}d + a\bar{d} + \bar{b}\bar{d}a + \bar{a}bde$	$\bar{b}c$	2

$SOP = ISOP = abc + acd + ad + bda + ec + abde + ed$			
Binate literal l_i	S_i	$ISOP - S_i$	$N_{BV}(S_i) + N_{BV}(ISOP-S_i)$
c	$ad + bda + abde + bc$	$abc + acd$	3
d	$abc + ad + bda + bc$	$acd + abde$	3
d	$abc + acd + ed + abde + bc$	$ad + bda$	3

Each row in Table 9 shows S_i and $ISOP - S_i$ obtained by *Subset*(*SOP*, l_i) for binate literals $l_i = a,a,b,b,c,c,d$ and d of *ISOP*. Row 3 (i.e. binate literal *b*) gives the minimum $N_{BV}(S_i) + N_{BV}(SOP - S_i)$ and so is selected. The selected S_i is still binate, so the second iteration of the outer loop is performed with the S_i assigned to *SOP*; see Table 10. Each row gives *Subset*(*SOP*, l_i) for binate literals $l_i = d$ and d in *SOP*.

TABLE 10

$SOP = S_{best} = ad + bda + ed + bc$			
Binate literal l_i	S_i	$ISOP - S_i$	$N_{BV}(S_i) + N_{BV}(ISOP-S_i)$
d	$ad + bda + bc$	$ed + abc + acd + abde$	1
d	$ed + bc$	$ad + bda + abc + acd + abde$	3

The first row (i.e. literal *d*) of Table 10 gives the lower $N_{BV}(S_i) + N_{BV}(SOP - S_i)$, and is selected. The selected S_i now is unate and so the process is done. We finally obtain $g_u = ad + bda + bc$ and $g_b = ed + abc + acd + abde$. Since *Find-Unate-Cube-Set*(*ISOP*) aims to reduce both $N_{BV}(S_i)$ and $N_{BV}(SOP - S_i)$, it tends to make g_b highly unate as well. Observe that the g_b produced in this example is unate for all but one variable (*a*).

The *Subset*(*SOP*, l_i) operation conducted using an *M*-node ZSBDD has a complexity of *O*(*M*). *Find-Unate-Cube-Set*(*ISOP*) for an *ISOP* with *N* binate

variables repeats the inner loop N^2 times. Hence the worst case complexity of *Find-Unate-Cube-Set(ISOP)* is $O(N^2M)$. Compare this with the complexity $O(2^m)$ of an exact method discussed above; m is usually significantly greater than N and M. Thus the presented AND-OR decomposition process can generate highly unate g_u^j and g_b^j quite efficiently.

While embodiments of the invention have been illustrated and described, it is not intended that these embodiments illustrate and describe all possible forms of the invention. Rather, the words used in the specification are words of description rather than limitation, and it is understood that various changes may be made without departing from the spirit and scope of the invention.